Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Mobilní OpenGL ES aplikace

Petr Navrátil

Bakalářská práce

2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2017/2018**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr Navrátil**
Osobní číslo:	**I15110**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Mobilní OpenGL ES aplikace**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Cílem práce je vytvořit ukázkovou mobilní aplikaci pro OS Android, která bude demonstrovat možnosti knihovny OpenGL ES.
V teoretické části bude stručně popsán OS Android a způsob tvorby mobilní aplikace pomocí vybraného IDE. Dále bude popsáno OpenGL ES API pro práci s 2D a 3D grafikou, včetně vysvětlení důležitých pojmů a popisu klíčových tříd a metod. V praktické části bude realizována demonstrační aplikace, která bude umět vykreslit 2D/3D objekty a využít techniky pro práci s texturami, se světelnými zdroji, kamerou a animacemi.

Rozsah grafických prací:

Rozsah pracovní zprávy: **min. 30 stran**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury:

BROTHALER, Kevin. OpenGL ES 2 for Android: a quick-start guide. Dallas, Texas: Pragmatic Bookshelf, 2013. ISBN 1937785343.
DAN GINSBURG, Budirijanto Purnomo a Aaftab Munshi. WITH EARLIER CONTRIBUTIONS FROM DAVE SHREINER. Opengl es 3.0 programming guide. Second edition. S.l.: Addison-Wesley, 2014. ISBN 0321933885.
GRIFFITHS, Dawn a David GRIFFITHS. Head first Android development. Sebastopol: O'Reilly, 2015. Head first series. ISBN 1449362184.

Vedoucí bakalářské práce: **Ing. Zdeněk Šilar, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **31. října 2017**

Termín odevzdání bakalářské práce: **12. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

Ing. Lukáš Čegan, Ph.D.
pověřený vedením katedry

V Pardubicích dne 20. března 2018

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 22. 11. 2017

Petr Navrátil

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu mé bakalářské práce panu Ing. Zdeňkovi Šilarovi, Ph.D. za odborné vedení a cenné rady, které mi poskytl v průběhu práce.

ANOTACE

V této práci je nejprve popsán Operační systém Android a následně je popsáno OpenGL ES API pro práci s 2D a 3D grafikou. Jsou také vysvětleny důležité pojmy a popsány klíčové třídy a metody. Následně je realizována demonstrační aplikace pro Operační systém Android, která umožňuje vykreslit 2D a 3D objekty a využívá techniky pro práci s texturami, světelnými zdroji, kamerou a animacemi.

KLÍČOVÁ SLOVA

Java, OpenGL ES, Android, mobilní aplikace, 2D/3D grafika

TITLE

OpenGL ES mobile application

ANNOTATION

In this work, the Android Operating System is first described, followed by the OpenGL ES API for 2D and 3D graphics. Key concepts and key classes and methods are also explained. Subsequently, a demonstration application for the Android Operating System is implemented, which allows to draw 2D and 3D objects and uses techniques for working with textures, light sources, camera and animations.

KEYWORDS

Java, OpenGL ES, Android, mobile application, graphics, 3D object

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK

API	Application Programming Interface
IDE	Integrated Development Environment
SDK	Software Development Kit
GUI	Graphical User Interface
AVD	Android Virtual Device
XML	Extensible Markup Language
JNI	Java Native Interface
GPU	Graphics Processing Unit
GLSL	OpenGL Shading Language
TMU	Texture Mapping Unit

ÚVOD

Cílem této práce je navrhnout a implementovat demonstrační mobilní aplikaci pro zařízení s operačním systémem Android, která bude demonstrovat jednotlivé možnosti práce s 2D a 3D grafikou s využitím technologie OpenGL ES. Práce nejdříve stručně popíše operační systém Android a také způsob tvorby mobilních aplikací pomocí IDE Android Studio. Dále bude popsáno OpenGL ES API pro práci s 2D a 3D grafikou. Součástí práce bude vysvětlení různých důležitých pojmů a budou popsány klíčové třídy a jejich metody, které se používají pro práci s grafikou při vývoji aplikace pro operační systém Android. V práci bude také stručně popsáno vývojové prostředí Android Studio, které se používá pro vývoj aplikací pro OS Android a které bylo také použito pro vytvoření demonstrační aplikace.

1 OPERAČNÍ SYSTÉM ANDROID

Operační systém (OS) Android je v současné době jeden z nejrozšířenějších operačních systémů využívaný mobilními zařízeními jako jsou chytré telefony, tablety, dále ho využívají také chytré televize a také různá tzv. nositelná elektronika. Tento operační systém je od svého počátku vyvíjen firmou Google, jeho první verze byla vydaná v roce 2008 a v současné době se nachází ve verzi číslo 8.1.0 (Obrázek 1).

Obrázek 1: Logo systému Android 8 „Oreo“

1.1 Architektura OS Android

Operační systém Android se skládá z několika hlavních komponent nebo také vrstev. Každá vrstva má na starosti několik věcí. V následujících kapitolách budou stručně popsány jednotlivé vrstvy [3].

1.1.1 Linux Kernel

Vrstva jádra operačního systému Linux. Tato vrstva je základní vrstvou platformy Android a mezi její funkcionality patří například správa vláken nebo nízko úrovňová správa paměti.

1.1.2 Hardware Abstraction Layer

Tato vrstva poskytuje standardní rozhraní, které poskytuje možnosti hardware vyšším vrstvám. Vrstva se skládá z několika knihovních modulů a každý modul implementuje rozhraní pro nějaký konkrétní typ hardware např. pro kameru nebo Bluetooth modul.

1.1.3 Android Runtime

Pro zařízení s operačním systémem Android verze 5.0 nebo vyšší každá aplikace má svůj vlastní proces, který běží v tzv. Android Runtime (ART). ART kompiluje kód v jazyce Java do kódu nativního, se kterým pracuje Linux kernel. Dříve se používal tzv. Dalvik virtual machine. ART přináší rychlejší běh aplikací, vylepšenou alokaci paměti, nové mechanismy správy paměti, nové možnosti ladění aplikací a přesnější vysokoúrovňové profilování aplikací.

1.1.4 Native C/C++ Libraries

Mnoho systémových komponent a služeb je vytvořeno v nativním kódu, který potřebuje nativní knihovny jazyka C/C++. Platforma Android poskytuje Java API Framework, který odkrývá a poskytuje funkcionality některých těchto nativních knihoven aplikacím. Příkladem je právě OpenGL ES. Android obsahuje API v jazyce Java, které využívá nativních knihoven a umožňuje nám kreslení a manipulaci s 2D a 3D grafikou v našich aplikacích. Vývojáři mohou také část aplikace vytvořit v nativním kódu a k tomu využijí také tyto knihovny.

1.1.5 Java API Framework

Sada všech možností operačního systému Android je dostupná přes API napsané v programovacím jazyce Java. Toto rozhraní obsahuje základní stavební prvky, moduly a komponenty, které jsou potřeba k tvorbě aplikací pro Android. Rozhraní je tvořeno množstvím balíčků, které obsahují třídy, rozhraní a metody v jazyce Java.

1.1.6 System Apps

Android obsahuje také několik základních aplikací, které můžou využívat jak běžní uživatelé, tak vývojáři, kteří můžou využívat některé funkcionality těchto aplikací ve své vlastní aplikaci.

2 VÝVOJ APLIKACÍ PRO PLATFORMU ANDROID

V této kapitole budou popsány možnosti a různé nástroje, které jsou potřeba k vývoji aplikací pro operační systém Android. Bude zde popsáno vývojové prostředí Android Studio, Android SDK a emulátor.

2.1 Android Studio

Když chceme vyvíjet aplikace, tak v první řadě potřebujeme něco, v čem budeme psát náš zdrojový kód. Pro tento účel můžeme použít Android Studio. Android Studio je tzv. integrované vývojové prostředí (IDE), které je v současné době asi nejpoužívanější vývojové prostředí pro vývoj aplikací pro operační systém Android.

Dříve se pro vývoj pro Android používalo vývojové prostředí Eclipse. Vývojové prostředí Eclipse bylo původně určeno k vývoji klasických desktopových aplikací v jazyce Java, ale postupně vznikly různé varianty a balíčky, které byly určeny pro vývoj různých typů programů a aplikací v různých programovacích jazycích. Jedním z těchto balíčků byl balíček, který se jmenoval Eclipse for Android Developers a byl určen právě pro vývoj aplikací pro Android. Poslední verze tohoto balíčku byla vydána 30.3.2016.

V prosinci 2014 bylo vydáno Android Studio ve verzi 1.0. Aktuální verze je verze 3.2, která byla vydána 26.3.2018. Toto vývojové prostředí je určeno přímo pro vývoj mobilních aplikací a je v současné době vyvíjeno firmou Google, která vyvíjí také operační systém Android. Základem pro toto prostředí je vývojové prostředí IntelliJ IDEA od firmy JetBrains. Android Studio obsahuje veškeré užitečné vývojářské nástroje pro vývoj aplikací pro Android. Obsahuje zvýrazňovač syntaxe, správce jednotlivých souborů v projektu, panel zobrazující strukturu aktuálního zdrojového souboru, zabudovaný terminál nebo třeba zabudovaný verzovací systém, který můžeme využít pro snadné zálohování do různých verzovacích systémů. Obsahuje správce SDK, který umožňuje jednoduše pomocí grafického rozhraní spravovat SDK, která jsou potřebná pro vývoj aplikace pro danou verzi operačního systému Android. Dále obsahuje správce virtuálních zařízení, umožňující vytvářet různé virtuální zařízení s různou velikostí obrazovky, vzhledem, velikostí paměti RAM, velikostí úložiště a dalšími vlastnostmi.

2.2 Android SDK

Nejdůležitějším požadavkem pro vývoj pro Android je tzv. Android Software Development Kit (SDK). Android SDK obsahuje knihovny a nástroje potřebné pro vývoj aplikací. Dále obsahuje různé nástroje pro tzv. debugování a testování programů nebo pro sestavování a kompilaci. Může také obsahovat dokumentaci, pro použití offline. SDK se liší podle toho pro jakou verzi operačního systému Android je určeno a jakou obsahuje verzi API neboli knihoven a tříd.

Verze SDK se odvíjí od toho, jaký typ aplikace chceme vytvářet, jaké bude obsahovat komponenty a pro jaké mobilní telefony bude určena. Čím vyšší verzi SDK použijeme, tím více různých komponent a tříd budeme mít k dispozici. Například, když použijeme SDK verze 23, tak tato verze odpovídá verzi operačního systému 6.0 neboli Marshmallow a znamená to, že naše aplikace bude fungovat na telefonech s operačním systémem verze 6.0 nebo vyšším.

2.3 Emulátor

Emulátor je program, který umožňuje virtualizaci fyzického zařízení. Pomocí emulátoru můžeme spouštět a testovat aplikace bez nutnosti mít fyzické zařízení. Lze ho využít k testování aplikací pro různé typy telefonů. Využívá k tomu virtuálních zařízení tzv. Android Virtual Devices (AVD), které se mohou lišit velikostí úhlopříčky displeje, operační paměti RAM, velikostí úložiště nebo dalšími různými parametry a můžeme tak vytvořit virtuální kopie různých konkrétních typů telefonů, které nemáme fyzicky k dispozici. Nevýhodou emulátoru je, že aplikace nejsou maximálně plynulé a také emulátor představuje značnou procesorovou a paměťovou zátěž pro náš počítač.

2.4 Základy vývoje aplikací pro Android

V této kapitole budou popsány základní pojmy a prvky používané při vývoji aplikací pro Android [2]. Pro tvorbu aplikací poskytuje OS Android čtyři základní komponenty, které jsou reprezentovány třídami `Activity`, `Service`, `Content Provider` a `Broadcast Receiver`. Nejzákladnější a nejjednodušší z těchto komponent je aktivita (activity), která bude popsána v samostatné kapitole a tu jsem také jako jedinou využíval v demonstrační aplikaci.

2.4.1 Aktivita

Aktivita představuje určitou konkrétní věc, kterou může uživatel v rámci naší aplikace provádět. Příkladem může být aktivita, která umožňuje uživateli psát emaily, nebo aktivita umožňující pořizování fotek. Aktivity jsou naprogramovány v jazyce Java a obvykle jedna aktivita představuje jednu obrazovku aplikace (není to pravidlem). Každá aktivita je nezávislá na ostatních aktivitách.

2.4.2 Intent

Intent představuje komunikační mechanismus, který slouží k přepínání aktivit, a protože jedna aktivita často představuje jednu obrazovku, tak také k přepínání obrazovek v naší aplikaci. Intenty jsou reprezentovány třídou `Intent`. Existují dva základní typy intentů: implicitní a explicitní. Při použití implicitního intentu nespecifikujeme cílovou aktivitu, ale pouze operaci, která má být vykonána. Operační systém sám vybere vhodnou aktivitu, která umí danou operaci vykonat. Pokud je vhodných aktivit více, uživatel vybere jednu z nich. Naproti tomu u explicitního intentu přímo specifikujeme konkrétní aktivitu, kterou chceme spustit. V demonstrační aplikaci se používají jen explicitní intenty, pro přepínání mezi jednotlivými scénami a úvodní obrazovkou.

2.4.3 Layout

Layouty popisují vzhled obrazovky, nebo také uživatelské rozhraní. Jsou napsány pomocí značkovacího jazyka XML a popisují, jak jednotlivé elementy na obrazovce vypadají, jaké mají vlastnosti a jak jsou na obrazovce rozmístěny. Jednotlivé layouty a komponenty, které jsou definované pomocí XML mají také vlastní třídy v jazyce Java a lze s nimi pracovat programově, a to i za běhu aplikace. Například lineární layout, který slouží k uspořádání elementů do jednoho řádku, nebo sloupce má třídu `LinearLayout`, nebo klasické tlačítko má třídu `Button`.

2.4.4 Soubor Android Manifest

Každá aplikace pro Android musí obsahovat soubor AndroidManifest.xml a tento soubor se musí nacházet v kořenovém adresáři projektu. Tento soubor obsahuje základní informace o aplikaci jako například seznam komponent, požadovaných knihoven, používaných senzorů

nebo oprávnění, které aplikace potřebuje. Při práci s OpenGL ES můžeme v tomto souboru specifikovat, že naše aplikace používá OpenGL ES určité verze a můžeme také určit, jestli je toto OpenGL nutný požadavek pro běh aplikace (Zdrojový kód 1). Pokud to specifikujeme jako nutný požadavek, potom uživatelům, jejichž zařízení nepodporuje požadované OpenGL nepůjde aplikace stáhnout z obchodu Google Play.

```
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true" />
```

Zdrojový kód 1: Specifikace požadované verze OpenGL ES

2.4.5 Zdroje

Každá aplikace používá méně nebo více různých zdrojů (*resources*). Základním zdrojem jsou XML soubory s layouty. Dalšími zdroji, které může aplikace používat mohou být různé obrázky, ikony, nadefinované styly, barvy, hodnoty a další. Zdroje se nachází ve složce `res` a v dalších podsložkách.

3 KNIHOVNA OPENGL ES

OpenGL je multiplatformní programovatelné rozhraní pro tvorbu 2D a 3D grafiky. OpenGL se používalo na klasických počítačích mnoho let a pro mobilní zařízení vzniklo tzv. OpenGL for Embedded Systems (OpenGL ES), které je určené pro různé mobilní zařízení jako jsou mobilní telefony, tablety, konzole a další.

OpenGL ES neobsahuje úplně všechny funkce klasického OpenGL, proto aby mohlo být implementováno na zařízeních s jednodušším, levnějším hardwarem, a hlavně aby mělo dostatečně nízké požadavky energii, tak aby mohlo fungovat na zařízeních, které využívají baterii.

3.1 Přehled verzí

OpenGL ES se stejně jako OpenGL postupně vyvíjí a rozšiřuje se o další pokročilé možnosti. Následuje stručný výčet a popis jednotlivých verzí:

OpenGL ES 1.0 – první verze, byla uvolněna pro veřejnost v roce 2003, vycházela z klasického OpenGL 1.3, mnoho funkcionality bylo ubráno a trochu také přidáno;

OpenGL ES 1.1 – tato verze přidala několik nových vlastností;

OpenGL ES 2.0 – verze, která byla uvolněna v roce 2007 přinesla mnoho zásadních změn, téměř všechny vlastnosti vykreslování, které byly v předchozích verzích specifikovány pomocí API s fixní množinou funkcí, byly nahrazeny tzv. shadery, které vytváří programátor, proto tato verze není zpětně kompatibilní s verzí 1.1;

OpenGL ES 3.0 – verze, jejíž specifikace byla dokončena v roce 2012 je plně kompatibilní s verzí 2.0 a rozšiřuje ji o několik pokročilých možností;

OpenGL ES 3.1 – tato verze byla vydána v roce 2014 a přidává několik nových možností;

OpenGL ES 3.2 – aktuální verze, která byla vydána v roce 2016 přidává zase další pokročilé možnosti a funkcionality.

V následujících částech této práce bude vždy popisováno OpenGL ES verze 2.0, které je pořád hojně rozšířené a které jsem použil při tvorbě demonstrační aplikace. Jedním z důvodů bylo také to, že Emulátor, který jsem používal při tvorbě a testování demonstrační aplikace nepodporuje OpenGL ES verze 3.0.

3.2 Základní pojmy používané při práci s grafikou a OpenGL ES

V této kapitole budou vysvětleny základní a další důležité pojmy, se kterými se setkáme ať už obecně při práci s jakoukoliv 2D nebo 3D grafikou, nebo se používají v souvislosti s OpenGL ES.

3.2.1 Vertex a Polygon

Vertex, česky vrchol představuje základní prvek každého 2D nebo 3D objektu. Vertex představuje bod ve dvojrozměrném nebo trojrozměrném prostoru. Jednotlivé objekty se skládají z vertexů, které jsou propojeny čárami a skupina několika propojených vertexů, které ohraničují určitý útvar tvoří *polygon* neboli mnohoúhelník. Všechny možné jednoduché nebo složitější objekty 2D nebo 3D světa jsou tvořeny skupinou polygonů a vertexů.

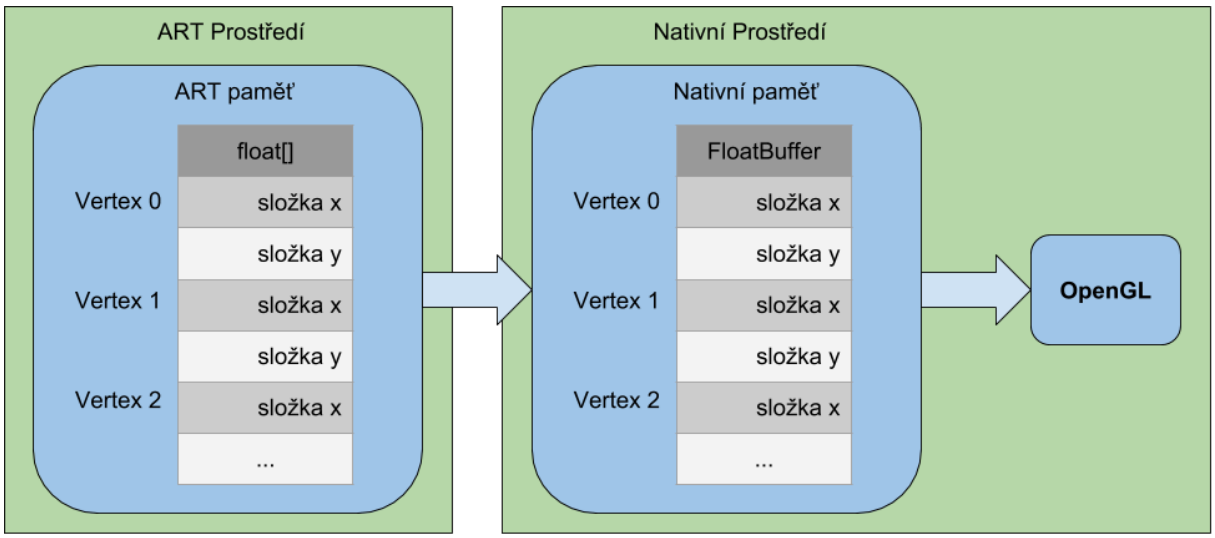
S pomocí OpenGL ES můžeme kreslit pouze body, čáry a trojúhelníky. Vertexy jsou definované pomocí čísel v plovoucí řádové čárce (datový typ `float` v jazyce Java). Vertex v trojrozměrném prostoru můžeme definovat pomocí tří těchto čísel.

3.2.2 Komunikace s OpenGL ES, nativní paměť a vertex buffer

Když nadefinujeme nějaký objekt pomocí vertexů musíme je zpřístupnit OpenGL. Problém je v tom, že prostředí, ve kterém běží náš kód není stejné jako prostředí kde běží OpenGL. Kód, který vytvoříme v jazyce Java neběží přímo na hardware našeho zařízení, ale běží ve speciálním virtuálním prostředí nazývaném Android Runtime (ART). Kód v tomto prostředí nemá žádný přímý přístup do nativního prostředí. Ke komunikaci s OpenGL použijeme tzv. Java Native Interface (JNI), ale toto rozhraní nepoužijeme přímo, protože to za nás řeší Android SDK. Pro komunikaci a práci s OpenGL budeme používat třídu `GLES20` z balíčku `android.opengl` a budeme používat její metody. Tato třída bude popsána v samostatné kapitole. Druhý problém, spočívá v tom, jak alokovat paměť. V jazyce Java máme několik speciálních tříd, které dokáží alokovat blok nativní paměti a zkopírovat do něj naše data. Tato nativní paměť bude přístupná nativnímu prostředí, ve kterém se nachází OpenGL (Obrázek 2).

Protože jsou vertexy popsány proměnnými typu `float`, tak pro alokaci nativní paměti budeme používat třídu `FloatBuffer`. Paměti, která slouží k ukládání pole vertexů, ve kterém jsou pro každý vertex definovány jednotlivé složky jeho pozice v prostoru a mohou zde být uloženy

další volitelné vlastnosti, které chceme pro vertexy uchovávat, tak této paměti se říká Vertex Buffer [1].



Obrázek 2: Komunikace s OpenGL ES

3.2.3 OpenGL pipeline, shadery, GLSL a rasterizace

Po definici struktury našeho objektu a překopírování dat do nativní paměti může začít OpenGL pracovat. Předtím než můžeme vykreslit náš objekt na obrazovku našeho zařízení, musíme ho poslat skrz tzv. *OpenGL pipeline* (Obrázek 3). OpenGL pipeline se skládá z malých programů, tzv. *shaderů*. Shadery říkají grafické kartě neboli grafické výpočetní jednotce (GPU) jak má vykreslit naše data. Existují dva základní typy shaderů a musíme oba definovat předtím, než můžeme vykreslit cokoliv na obrazovku [1].

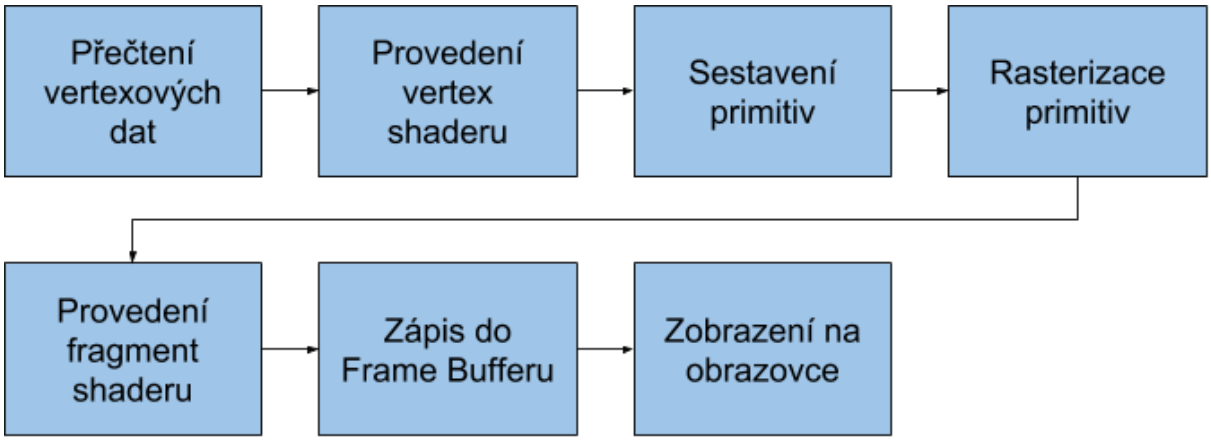
Vertex shader

Vertex shader generuje výslednou pozici vertexu a provádí se jednou pro každý vertex. Potom co jsou známy výsledné pozice, OpenGL vezme viditelnou množinu vertexů a sestaví z nich body, čáry a trojúhelníky.

Fragment shader

Fragment shader generuje výslednou barvu každého fragmentu bodu, úsečky nebo trojúhelníku a provádí se jednou pro každý fragment. Fragment je malý obdélníkový tvar jedné barvy, analogický pixelu na počítačové obrazovce.

Potom co jsou vygenerovány výsledné barvy, OpenGL je zapíše do bloku paměti, které se říká *frame buffer* a Android potom zobrazí obsah tohoto bufferu na obrazovce.



Obrázek 3: OpenGL ES pipeline

Ve verzi OpenGL ES 1.0 a 1.1 neexistovaly shadery a OpenGL používalo fixní sadu funkcí, která nám umožňovala kontrolovat několik málo věcí, jako třeba kolik bude světel ve scéně, nebo kolik mlhy přidat do scény. Výhodou tohoto API byla jednoduchost použití, ale na druhou stranu bylo složité na různé rozšíření. Pokud jsme chtěli například vytvořit různé vlastní efekty, většinou jsme měli smůlu. Jak se postupně vylepšoval hardware naších zařízení, tak OpenGL ES 2.0 přidalo programovatelné rozhraní využívající shadery a aby zůstaly věci jednoduché bylo odstraněno fixní API z dřívějších verzí a musí se používat shadery. Pomocí shaderů můžeme řídit, jak se každý vertex vykreslí na obrazovku a také jak se každý fragment každého bodu, čáry a trojúhelníku vykreslí. Můžeme tedy nyní vytvářet libovolné vlastní efekty, pokud je dokážeme naprogramovat pomocí jazyka pro programování shaderů, který se nazývá OpenGL ES Shading Language (GLSL).

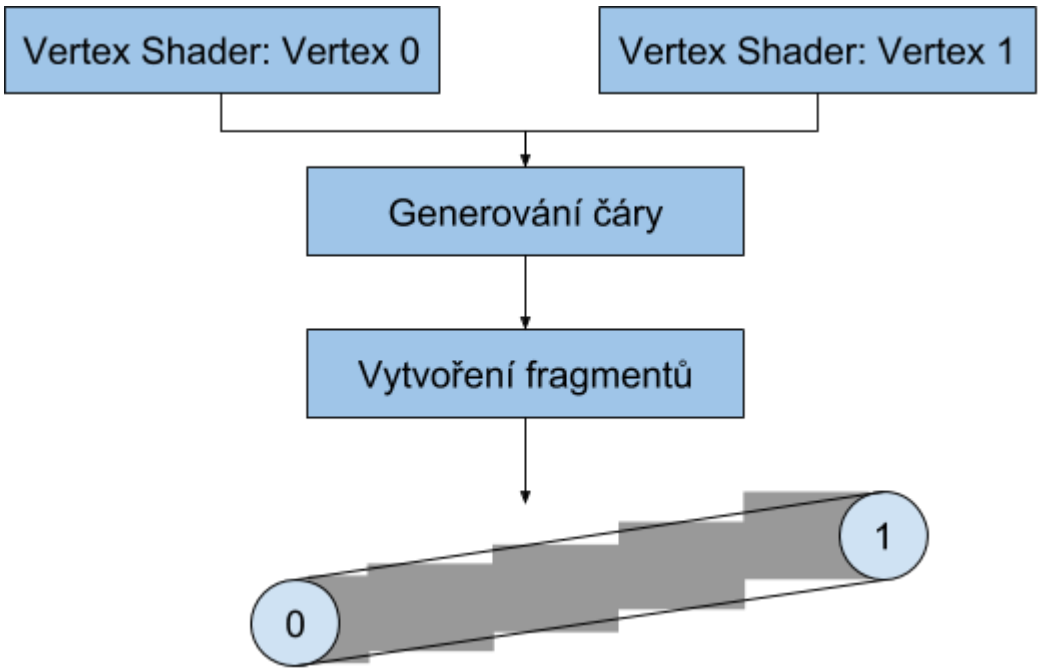
GLSL

GLSL je programovací jazyk, který má strukturu a syntaxi podobnou jazyku C. Pomocí tohoto jazyka, můžeme různými způsoby manipulovat s vertexy a fragmenty ve vertex nebo fragment shaderu. Umožňuje vytvářet vlastní funkce, které můžeme volat v rámci zpracování ve funkci `main()`, která je hlavní funkcí každého shaderu. Mezi nejpoužívanější datové typy patří datové typy pro desetinná čísla, vektory a matice, například datové typy `float`, `vec3`, `mat4`, nebo pro práci s texturami slouží datový typ `sampler2D`. Jazyk obsahuje několik speciálních zabudovaných proměnných, například `gl_Position`, která představuje výslednou pozici vertexu a musí být nastavena v rámci vertex shaderu, nebo proměnná `gl_FragColor`, která představuje výslednou barvu fragmentu a musí být nastavena v rámci fragment shaderu. GLSL obsahuje ještě několik speciálních typů proměnných například `uniform`, `attribute` nebo

`varying`. Proměnná typu `uniform` představuje konstantu, kterou nastavíme a její hodnota je pak pro všechny zpracovávané vertexy nebo fragmenty stejná. Typ `attribute` představuje vlastnost každého vertexu a hodnoty této proměnné jsou pro každý zpracovávaný vertex jiné a jsou načítány z vertex bufferu. Typ `varying` specifikuje proměnnou, která bude pro každý fragment jiná. Takováto proměnná musí být shodně definována jak ve vertex shaderu, tak ve fragment shaderu. Hodnota proměnné je námi požadovaným způsobem nastavena ve vertex shaderu a potom je předána do fragment shaderu, kde s ní můžeme dále pracovat. Všechny tyto popsané proměnné a typy proměnných byly použity v demonstrační aplikaci. Jazyk GLSL je značně složitější a v rámci této práce nebude podrobněji popsán.

Rasterizace

Mobilní displej se skládá z milionů pixelů a každý z těchto pixelů je schopen zobrazit jednu barvu. OpenGL vytváří obraz, který můžeme zobrazit pomocí pixelů naší obrazovky tak, že rozdělí každý bod, čáru nebo trojúhelník na skupinu malých fragmentů v průběhu procesu, který se nazývá *rasterizace*. Fragmenty jsou analogické pixelům a každý fragment obsahuje jednu barvu. K reprezentaci barvy má každý fragment čtyři složky: červenou, zelenou a modrou složku pro barvu a dále také tzv. alfa složku, která reprezentuje průhlednost. S těmito barevnými složkami a dalšími věcmi můžeme pracovat v rámci fragment shaderu. Na obrázku (Obrázek 4) je znázorněna rasterizace čáry.



Obrázek 4: Rasterizace čáry

3.2.4 Textury

Textury jsou obrázky, které se nahrají do OpenGL a můžou být vykresleny na povrchy různých objektů místo klasické jednoduché barvy. Jádrem každé hry nebo 3D scény jsou pouze objekty tvořené pomocí bodů, čar a trojúhelníků a jednotlivé povrchy těchto objektů jsou otexturovány a tak vznikají krásné 3D scény. Textury obecně umožnují přidat do naší scény velké množství detailů. Textury jsou tvořeny pomocí malých bodů tzv. texelů, které jsou analogické fragmentům a pixelům. Nejběžnější způsob použití textur je načtení dat přímo ze souboru s obrázkem. Každá textura má svůj souřadnicový systém, jehož souřadnice začínají v jednom rohu textury hodnotami `(0,0)` a končí v protilehlém rohu textury hodnotami `(1,1)`. Podle konvencí se jeden rozměr označuje jako S a druhý rozměr jako T.

V OpenGL ES 2.0 nemusí být textury čtvercové, ale každý rozměr musí být v mocninách čísla dvě. Textury jejichž rozměry nejsou v mocninách čísla dvě mají velmi omezené použití, naproti tomu textury s rozměry v mocninách můžou být použity ve všech případech. Existuje také maximální velikost textury, která se liší podle implementace OpenGL ES, ale většinou to je hodnota 2048*2048 bodů.

3.2.5 Používání transformačních matic v OpenGL

Každá 2D nebo 3D aplikace sdílí jeden velký problém, jak se rozhodnout co se zobrazí na displeji a jak to přizpůsobit rozměrům displeje. v OpenGL můžeme použít tzv. projekci k zobrazení časti našeho světa na obrazovku a můžeme to udělat tak, že to vypadat správně napříč různými velikostmi a orientacemi obrazovky. Všechno, co vykreslíme pomocí OpenGL je v rozsahu souřadnic `[-1,1]` na osách x a y, případně ještě na ose z, pokud pracujeme ve 3D. Souřadnice v tomto rozsahu se nazývají jako normalizované souřadnice zařízení (anglicky *normalized device coordinates*) a jsou nezávislé od aktuální velikosti nebo tvaru obrazovky. Pokud bychom tyto souřadnice používali přímo, tak by naše scéna mohla vypadat různě na různých zařízeních. Potřebujeme upravit souřadnicový systém tak, aby bral v potaz velikost obrazovky a poměr stran. Nesmíme pracovat přímo s normalizovaným souřadnicovým systémem, ale musíme začít pracovat s virtuálním souřadnicovým systémem (anglicky *virtual coordinate space*) a musíme mít způsob, jak převádět souřadnice z virtuálního souřadnicového systému zpátky do normalizovaného. K transformaci virtuálních souřadnic zpátky na normalizované se používá tzv. projekce.

Mnoho věcí v OpenGL pracuje s maticemi a vektory a jeden z nejdůležitějších případů použití matic je vytvoření ortografické a perspektivní projekce. Souřadnice jednotlivých vertexů, které tvoří objekty naší scény jsou uloženy ve formě vektorů a pomocí projekčních matic můžeme souřadnice těchto vertexů různě měnit, nebo transformovat. Tyto transformace se provádějí pomocí několika operací násobení nebo sčítání matic s vektory. Například pro projekci nějakého objektu musíme všechny vertexy tohoto objektu vynásobit projekční maticí.

Ortografická projekce

Při použití této projekce všechno vypadá stejně velké bez ohledu na to jak blízko nebo daleko se to nachází. Tato projekce se může použít u 2D scén a v demonstrační aplikaci je využívána ve scéně 1.

Perspektivní projekce

Projekce, která se používá pro vytvoření iluze trojrozměrného prostoru na obrazovce našeho zařízení. Objekty, které se nacházejí dál jsou menší a ty které jsou blíž k pozorovateli jsou naopak větší. Tato projekce je v demonstrační aplikaci využívána ve scéně 2 a ve scéně 3.

Dalšími používanými maticemi jsou například matice pro posun objektu daným směrem (*translation matrix*), matice pro rotaci objektu (*rotation matrix*) nebo matice pro změnu velikosti objektu. Při práci ve 3D se používá kromě složek x, y a z ještě složka w, která se používá k reprezentaci vzdálenosti. Následuje stručný popis toho, jak je složka w používána.

Clip Space

Když vertex shader zapíše hodnotu (pozici vertexu) do speciální proměnné `gl_Position`, OpenGL očekává, že tato pozice se bude nacházet v tzv. *clip space*. Clip space funguje velmi jednoduše: pro každou zadanou pozici se musí složky x, y a z nacházet v intervalu `[-w,w]` pro danou pozici. Cokoliv mimo tento interval nebude viditelné na obrazovce.

Perspektivní rozdělení

Předtím než z pozice vertexu vzniknou normalizované souřadnice zařízení, OpenGL provede ještě jeden krok, který se nazývá perspektivní rozdělení (*perspective division*). Po provedení této operace budou pozice v normalizovaných souřadnicích zařízení, kde každá viditelná souřadnice má hodnoty složek x, y a z v intervalu `[-1,1]`. Pro vytvoření iluze 3D OpenGL vezme každou složku x, y a z vydělí ji složkou w. Protože složka w představuje vzdálenost, způsobí to, že vzdálené objekty budou přesunuty blíže ke středu vykreslovací oblasti, který chová jako úběžník. Tímto způsobem vniká iluze 3D.

V této kapitole byly popsány různé základní a důležité pojmy, týkající se grafiky a OpenGL ES a v následujících kapitolách budou popsány konkrétní třídy a prostředky používané při programování v jazyce Java.

3.3 Balíček `android.opengl`

Balíček `android.opengl` obsahuje všechny třídy a rozhraní používané při práci s OpenGL ES. Jádrem tohoto balíčku jsou třídy, které zabalují nativní rozhraní, které slouží ke komunikaci a samotné práci s OpenGL ES dané verze, tyto třídy jsem jmenují vždy GLES plus číslo verze OpenGL ES se kterou pracují. Například třída `GLES10` tvoří rozhraní pro práci s OpenGL ES ve verzi 1.0, nebo třída `GLES32` pro práci s OpenGL ES ve verzi 3.2. Třída `GLES20` se kterou jsem pracoval v rámci ukázkové aplikace bude podrobněji popsána v samostatné kapitole. Všechny tyto třídy jsou statické a obsahují mnoho statických metod, které slouží k volání jednotlivých funkcí OpenGL ES a dále obsahují mnoho statických celočíselných konstant, které jsou v různých případech používány jako parametry při volání statických metod [6].

Dále balíček obsahuje třídy a rozhraní pro zobrazení a vykreslování grafiky uvnitř naší aplikace, respektive aktivity, tyto třídy právě využívají OpenGL ES. Příkladem je třída `GLSurfaceView`. Tyto třídy budou podrobněji popsány v samostatné kapitole.

Balíček také obsahuje několik pomocných tříd, které obsahují zejména různé statické metody pro zjednodušení určitých například často se opakujících operací, které by jinak musel programátor vytvářet nebo řešit sám. Mezi tyto třídy patří například třída `GLUtils`, nebo třída `Matrix`, která obsahuje různé metody určené pro provádění různých matematických operací s maticemi a vektory.

Nakonec je v balíčku ještě několik dalších tříd a rozhraní, které ale nepotřebujeme při základní práci s OpenGL využívat a jsou využitelné při tvorbě nějakých pokročilejších věcí.

3.3.1 Třída `GLSurfaceView`

Tato třída představuje speciální komponentu, na kterou můžeme kreslit OpenGL ES grafiku. Spravuje tzv. surface, na který může OpenGL kreslit. Tato třída využívá programátorem poskytnutý tzv. Renderer, který provádí vlastní vykreslování. Pro vykreslování je vytvořeno samostatné vlákno, aby se oddělilo provádění vykreslování od vlákna, které má na starost uživatelské rozhraní. Třída zjednodušuje řešení standardního životního cyklu aktivity. Aktivita

může být vytvořena, zničena, pozastavena, nebo znovu puštěna, když třeba uživatel přejde do jiné aktivity a potom se vrátí zpět a při přechodech mezi těmito stavy musíme správně uvolňovat různé zdroje, které používá OpenGL. `GLSurfaceView` poskytuje pomocné metody, které tyto věci řeší za nás. Tato třída podporuje plynulé vykreslování a také vykreslování na vyžádání. Může také volitelně testovat, sledovat a zaznamenávat chyby jednotlivých OpenGL volání prováděných rendererem.

Použití této třídy je doporučeno a typicky se provádí vytvořením potomka, pokud potřebujeme nějakým způsobem reagovat na různé typy událostí, jako například když se uživatel dotkne plochy, na kterou se vykresluje. Nicméně lze použít také přímo tuto třídu a není nutné vytvářet žádného potomka. Tato třída sama neprovádí vykreslování, ale využívá renderer a pro jeho přiřazení se používá metoda `setRenderer()`. Další důležitá metoda je metoda `setEGLContextClientVersion()`, pomocí které nastavujeme, kterou verzi OpenGL chceme používat. Zavoláním této metody s parametrem 2 nastavíme náš surface view tak aby používal OpenGL ES 2.0.

3.3.2 Rozhraní `GLSurfaceView.`*`Renderer`*

Toto rozhraní je zodpovědné za vytvoření OpenGL volání, které budou zavolány při vykreslení snímku. Typicky se toto rozhraní používá tak, že vytvoříme vlastní třídu, která implementuje toto rozhraní a tato třída je následně použita jako renderer ve třídě `GLSurfaceView`, případně v jejím námi vytvořeném potomkovi. Rozhraní obsahuje tři metody, které musíme implementovat [1].

`onSurfaceCreated()`

`GLSurfaceView` volá tuto metodu, když je vytvořen povrch pro kreslení. to se stává při prvním spuštění naší aplikace a také při probuzení zařízení z režimu spánku, nebo když uživatel přepne zpátky do této aktivity z nějaké jiné aktivity. Znamená to, že tato metoda může být zavolána mnohokrát během běhu naší aplikace.

`onSurfaceChanged()`

Tato metoda je zavolána potom co je povrch pro kreslení vytvořen a také kdykoliv, když je změněna jeho velikost. Změna velikosti povrchu určeného pro vykreslování nastává nejčastěji při změně orientace našeho zařízení z tzv. Portrait módu do Landscape módu, nebo obráceně.

`onDrawFrame()`

`GLSurfaceView` volá tuto metodu, když je čas vykreslit nový snímek. Vždy musíme něco vykreslit, i když je to jen třeba vyčištění obrazovky. Toto je nejdůležitější metoda tohoto rozhraní. V této metodě voláme různé metody s jejichž pomocí provádíme vykreslování námi požadovaných objektů, efektů a dalších věcí.

Parametry těchto metod jsou při používání OpenGL ES verze 2.0 nebo vyšší prakticky nepoužívané, používali se zejména dříve při práci s OpenGL ES verze 1.0.

3.3.3 Třída `GLES20`

Tato třída představuje rozhraní pro komunikaci a práci s OpenGL ES verze 2.0. V této kapitole budou postupně popsány důležité metody používané pro tvorbu různé grafiky pomocí OpenGL ES [1].

Základní metody

`glClearColor()` – slouží k nastavení barvy, kterou se vyčistí povrch před každým dalším zavoláním metody `onDrawFrame()`, parametrem metody jsou jednotlivé barvené složky pro nastavení požadované barvy;

`glClear()` – tato metoda bývá většinou prvním příkazem uvnitř metody `onDrawFrame()` a pokud má jako parametr konstantu `GL_COLOR_BUFFER_BIT`, tak způsobuje vyplnění celého kreslicího povrchu barvou, kterou jsem nastavili pomocí `glClearColor()`;

`glViewport()` – pomocí této metody nastavíme velikost povrchu, tzv. viewport, který má OpenGL dostupný pro vykreslování grafiky. Většinou tento viewport přímo odpovídá velikosti `GLSurfaceView`, takže se využívá se povrch pro kreslení.

Metody pro vytváření, načtení a používání shaderů

`glCreateShader()` – vytváří objekt shaderu a parametrem této metody je buď konstanta `GL_VERTEX_SHADER` pro vytvoření vertex shaderu, nebo `GL_FRAGMENT_SHADER` pro fragment shader. Metoda vrací identifikátor (ID), nebo také referenci na tento objekt. ID je celé číslo typu `int` a toto číslo by bylo možné přirovnat ukazatelům v jazyku C. S těmito celočíselnými identifikátory pracuje většina metod ve třídě `GLES20` a budou se mnohokrát v dalších částech a kapitolách vyskytovat. Kdykoliv, když budeme chtít někde s tímto objektem pracovat, tak předáme jeho ID jako parametr při volání nějaké konkrétní metody třídy `GLES20`. Pokud vytvoření objektu selže metoda vrací hodnotu 0;

`glShaderSource()` – touto metodou přiřadíme existujícímu objektu shaderu vlastní kód tohoto shaderu. Kód shaderu představuje kód v jazyce GLSL, který definuje samotnou funkcionalitu tohoto shaderu, co má shader dělat. Prvním parametrem metody je ID shaderu a druhým kód shaderu ve formátu textového řetězce;

`glCompileShader()` – po vytvoření objektu shaderu a vložení kódu použijeme tuto metodu k tomu abychom zkompilovali kód shaderu. Parametrem metody je ID shaderu, který chceme zkompilovat;

`glGetShaderInfoLog()` – touto metodou můžeme získat chybovou zprávu pokud se nepovede kompilace shaderu. Pokud se kompilace nepovede a OpenGL má pro nás nějakou informaci, nebo zprávu proč se kompilace nepovedla, tak jí uloží do informačního logu daného shaderu. Parametrem metody je ID shaderu, jehož informační zprávu chceme získat a návratová hodnota metody je textový řetězec obsahující tuto zprávu;

`glDeleteShader()` – slouží pro zrušení existující objektu shaderu. Parametrem je ID objektu, který chceme zrušit. Tuto metodu můžeme použít třeba po neúspěšné kompilaci;

`glCreateProgram()` – pomocí této metody vytvoříme tzv. OpenGL program. OpenGL program je tvořen jedním vertex shaderem a jedním fragment shaderem, které jsou spojeny do jednoho objektu. Návratovou hodnotou metody je ID nově vytvořeného OpenGL programu;

`glAttachShader()` – touto metodou připojíme existující objekt shaderu k existujícímu objektu OpenGL programu. Typicky tuto metodu voláme dvakrát, jednou připojíme vertex shader k danému programu a napodruhé připojíme fragment shader k tomu samému programu. Prvním parametrem metody je ID programu, druhým parametrem je ID připojovaného shaderu;

`glLinkProgram()` – když už máme připojeny shadery k programu, musíme je ještě spojit, nebo svázat. Parametrem je ID programu, jehož shadery chceme spojit;

`glUseProgram()` – tato metoda slouží k nastavení aktuálního programu, který bude OpenGL používat při vykreslování čehokoliv na obrazovku. Při vykreslování se používá vertex a fragment shader aktuálního OpenGL programu. Parametrem je ID programu, který chceme nastavit jako aktivní.

Metody pro práci se shadery

`glGetUniformLocation()` – při práci s shadery můžeme mít v zdrojovém kódu shaderu nadefinovány různé proměnné, konstanty a atributy. Hodnoty konstant jsou pro všechny vertexy zpracovávané shaderem stejné. Tato metoda slouží k získání ID požadované konstanty,

které budeme používat, když budeme chtít například nastavit hodnotu dané konstanty. Prvním parametrem je ID OpenGL programu a druhým parametrem je textový řetězec představující název konstanty jejíž ID chceme získat. Zadaný název konstanty musí přesně odpovídat názvu konstanty, která je definována ve zdrojovém kódu daného shaderu popsaném v jazyce GLSL;

`glGetAttribLocation()` – tato metoda funguje obdobně jako metoda `glGetUniformLocation()`, akorát slouží k získání ID atributu. Atributy shaderů představují takovou proměnnou, jejíž hodnota je různá pro každý zpracovávaný vertex. Klasickým příkladem atributu je například pozice v prostoru. Každý námi definovaný vertex má jinou pozici, a proto každý vertex v rámci zpracování v průběhu OpenGL pipeline má svoje vlastní hodnoty atributů;

`glVertexAttribPointer()` – v kapitole 3.2.2 bylo popsáno, že pro zpřístupnění dat OpenGL musíme vložit data do nativní paměti a k tomu použijeme třídu `FloatBuffer`. Do tohoto bufferu ukládáme jednotlivé hodnoty atributů všech nadefinovaných vertexů. Pomocí metody `glVertexAttribPointer()` říkáme OpenGL kde nalezne data jednotlivých vertexů pro daný atribut. Prvním parametrem metody je ID atributu, pro který chceme nastavit kde OpenGL najde data. Druhým parametrem je počet dat nebo hodnot pro jeden atribut. Například když bychom měli nadefinovány vertexy pro použití ve 2D prostoru a každý vertex by měl tedy hodnotu x a hodnotu y, tak tento parametr nastavíme na hodnotu 2, protože každý vertex bude při zpracování ve vertex shaderu potřebovat mít v atributu vloženy 2 hodnoty. Třetím parametrem je typ dat. Když budeme pro hodnoty používat datový typ `float`, tak jako tento parametr nastavíme konstantu `GL_FLOAT`. Čtvrtý parametr se využívá jen pokud jako data používáme celá čísla. Většinou tento parametr můžeme ignorovat. Pátý parametr se využívá, pokud do jednoho bufferu ukládáme více než jeden atribut. Například bychom mohli mít pro každý vertex uloženy hodnoty pro souřadnice a také hodnoty pro barevné složky. Tímto parametrem nastavíme po kolika hodnotách začínají hodnoty pro další vertex, takže pokud pro souřadnice jsou například 2 hodnoty a pro barevné složky 4 hodnoty, tak tento parametr nastavíme na hodnotu 6. Posledním parametrem metody je buffer, ze kterého má OpenGL číst data. Tato metoda je jedna z nejdůležitějších a musíme si dávat pozor, protože při nastavení špatných parametrů bychom mohli dostával zvláštní výsledky při vykreslování, dokonce by to mohlo způsobit pád programu;

`glEnableVertexAttribArray()` - potom co nastavíme zdroj dat pro atribut musíme tento atribut pomocí této metody aktivovat. Parametrem je ID atributu, který chceme aktivovat.

Metody pro nastavování konstant ve shaderech

Další množství metod ve třídě `GLES20` slouží k nastavování hodnot konstantám definovaným ve shaderech. Tyto metody se jmenují podle toho, jaký typ konstanty nastavují. V jejich názvech je vždy specifikován typ konstanty a také se podle daného typu liší jejich parametry. První parametr každé z těchto metod je ID konstanty, které chceme nastavit hodnotu. Následují některé příklady těchto metod;

`glUniform1i()` – tato metoda slouží k nastavení konstanty, která je typu `int`, proto má v názvu písmeno „i“ a je tvořena jedinou hodnotou, proto v názvu „1“. Parametrem je hodnota typu `int`, kterou chceme nastavit;

`glUniform4f()` – pomocí této metody nastavíme hodnotu konstanty, která je v shaderu typu `vec4`, což znamená, že se jedná o čtyřrozměrný vektor a protože v názvu metody je písmeno „f“ tak složky tohoto vektoru jsou typu `float`. Parametry metody jsou čtyři složky, které chceme nastavit danému vektoru;

`glUniformMatrix4fv()` – tato metoda slouží k nastavení konstanty typu matice o čtyřech řádcích a čtyřech sloupcích, jejíž hodnoty jsou typu `float`. Je potřeba upozornit na to, že v OpenGL jsou matice reprezentovány pomocí jednorozměrných polí a jsou sloupcově orientované, takže jsou v těchto polích za sebou uloženy jednotlivé sloupce. Pro reprezentaci matice 4x4 tedy potřebujeme jednorozměrné pole o velikosti 16 prvků. Předposlední parametr této metody představuje právě toto pole, pomocí kterého nastavíme jednotlivé hodnoty matice.

Metody pro vykreslování

OpenGL obsahuje několik metod pro vykreslování. Tyto metody provádějí samotné kreslení námi definovaných objektů a grafiky. V následující části budou postupně popsány tyto metody.

`glDrawArrays()` – toto je základní metoda pro kreslení objektů. Metoda vykresluje vertexy uložené ve vertex bufferu neboli nativní paměti. Před jejím zavoláním musíme už mít zavolány metody `glVertexAttribPointer()` a `glEnableVertexAttribArray()`, pomocí kterých nastavujeme a aktivujeme „zdroj dat“ se kterými bude později pracovat právě metoda `glDrawArrays()`. Tato metoda má 3 důležité parametry. Prvním parametrem je typ objektů, které chceme vykreslovat a těchto typů je několik. Prvním typem objektu je bod a pro něj použijeme jako parametr konstantu `GL_POINTS`. Pro vykreslení bodu nám stačí mít jeden vertex ve vertex bufferu. Jako další můžeme vykreslit čáru, pro kterou použijeme jako parametr konstantu `GL_LINES`. Pro vykreslení čáry potřebujeme dva vertexy. Dalším a asi

nejpoužívanějším typem objektu pro vykreslování je trojúhelník. Pro něj použijeme konstantu `GL_TRIANGLES` a potřebujeme tři vertexy pro jeden trojúhelník. Toto jsou tři základní objekty, které můžeme vykreslit. Existuje ještě několik dalších objektů, například objekt, který se anglicky nazývá „*triangle fan*“ (konstanta `GL_TRIANGLE_FAN`), nebo objekt „*triangle strip*“ (konstanta `GL_TRIANGLE_STRIP`), a oba tyto objekty jsem použil v rámci demonstrační aplikace. Druhý parametr metody udává, z jaké pozice ve vertex bufferu chceme začít číst vertexy pro vykreslení. Když chceme číst od začátku, použijeme hodnotu 0. Poslední parametr udává, kolik vertexů má OpenGL přečíst. Například, když bychom chtěli nakreslit 6 trojúhelníků, tak hodnotu třetího parametru bychom nastavili na 18, protože trojúhelník se skládá ze 3 vertexů a 3*6 je 18;

`glDrawElements()` – použití této metody je podobné jako metody `glDrawArrays()`, ale tato metoda využívá tzv. indexů. Indexy lze použít, když potřebuje vykreslit třeba několik různých trojúhelníků, které mají některý z vertexů společný. Místo toho abychom měli definováno několik stejných vertexů, tak použijeme indexové pole, kde pro každý trojúhelník specifikujeme, z jakých vertexů ve vertex bufferu se skládá a pak nemusíme mít stejné vertexy definované víckrát.

Metody pro práci s texturami

`glGenTextures()` – tato metoda slouží k vytvoření objektu textury a funguje podobně jako metody `glCreateShader()` nebo `glCreateProgram()`. Prvním parametr udává, kolik ID textur chceme vytvořit. Typicky se používá hodnota 1. Druhým parametrem je reference na pole, do které se vytvořené ID uloží. Třetí parametr udává offset od začátku pole, pro nastavení pozice v poli, od které se budou ID ukládat;

`glDeleteTextures()` – pokud chceme zrušit objekty textury použijeme tuto metodu. Paramery jsou obdobné jako u metody `glGenTextures()`, ale první parametr udává kolik objektů chceme zrušit a druhý parametr představuje pole objektů (ID) textur, které chceme zrušit;

`glBindTexture()` – touto metodou provedeme provázání textury s OpenGL. To znamená, že různé další metody pro práci s texturami budou pracovat s touto provázanou texturou. Prvním parametrem je typ textury, pro který se používá konstanta `GL_TEXTURE_2D` a druhým parametrem je ID textury, kterou chceme provázat. Je dobrým zvykem po provedení požadovaných operací s texturou texturu odvázat pomocí volání

`glBindTexture(GL_TEXTURE_2D, 0)`, abychom si omylem tuto texturu nějakým způsobem nezměnili při volání dalších metod;

`glActiveTexture()` – při kreslení s texturami neposíláme textury přímo do shaderu jako jiné proměnné nebo konstanty. Místo toho používáme tzv. texture mapping unit (TMU), česky texturovací jednotku. Texturovací jednotka je jednou ze součástí grafické karty a slouží k uchovávání textur, které jsou aktuálně vykreslovány. Můžeme používat více texturovacích jednotek pro vykreslování více než jedné textury současně. Metoda `glActiveTexture()` slouží k nastavení aktivní texturovací jednotky. Parametrem metody je konstanta představující číslo nebo ID texturovací jednotky. Pro nastavení nulté texturovací jednotky jako aktivní použijeme parametr `GL_TEXTURE0`. Počet texturovacích jednotech v OpenGL ES ve verzi 2.0 je 32. Poslední jednotku bychom nastavili pomocí parametru `GL_TEXTURE31`.

Pro práci s texturami se používá ještě jedna metoda, která není součástí třídy `GLES20`. Je to metoda **`texImage2D()`** ze třídy `GLUtils`. Tato metoda slouží k přečtení obrazových dat uložených v objektu typu Bitmap, které jsme dříve načetli ze souboru (z nějakého obrázku) a jejich překopírování do objektu textury, který je aktuálně svázán s OpenGL. Nejdůležitějším parametrem této metody je třetí parametr, který představuje referenci na objekt s obrazovými daty, které chceme použít jako texturu.

3.3.4 Třída `Matrix`

Tato třída obsahuje statické metody, které umožňují snadné vytváření a nastavování různých typů transformačních matic a také několik metod které slouží například k násobení dvou matic, násobení matice a vektoru, nebo k výpočtu velikosti vektoru. Následuje popis některých z těchto metod.

`multiplyMM()` – tato metoda slouží k vynásobení dvou matic. Prvním parametrem je pole, do kterého bude uložena výsledná matice, druhým parametrem je offset do této matice udávají kam se má uložit výsledná matice. Třetím parametrem je první matice a čtvrtým parametrem její offset a poslední dva parametry jsou druhá matice a její offset;

`length()` – metoda sloužící k výpočtu velikosti trojrozměrného vektoru. Jejími parametry jsou tři čísla typu `float` představující jednotlivé složky vektoru;

`orthoM()` – tato metoda slouží k nastavení matice pro ortografickou projekci. Jejími parametry jsou výsledná matice a potom číselné hodnoty představující jednotlivé hranice

obrazovky. Ve směru osy X je to levá a pravá hranice, ve směru osy Y horní a dolní hranice a ve směru osy Z přední a zadní hranice;

`perspectiveM()` – metoda, která slouží k nastavení matice pro perspektivní projekci. Parametry jsou: vertikální zorný úhel, poměr stran a hraniční souřadnice na ose Z.

4 IMPLEMENTACE UKÁZKOVÉ MOBILNÍ APLIKACE

Součástí této práce bylo vytvořit demonstrační mobilní aplikaci, která bude umět vykreslovat 2D a 3D objekty, bude využívat techniky pro práci s texturami, světelnými zdroji, kamerou a animacemi. Jak už bylo zmíněno v předchozí části práce, aplikace využívá OpenGL ES verze 2.0, protože verze 3.0 je zpětně kompatibilní a přidává jen několik další možností a také protože emulátor nepodporuje verzi 3.0. V následující části budou popsány jednotlivé prvky a komponenty této aplikace. Nejdříve budou popsány společné prvky, komponenty a různé třídy využívané ve všech scénách a následně budou podrobněji popsány jednotlivé scény.

4.1 Základní struktura aplikace

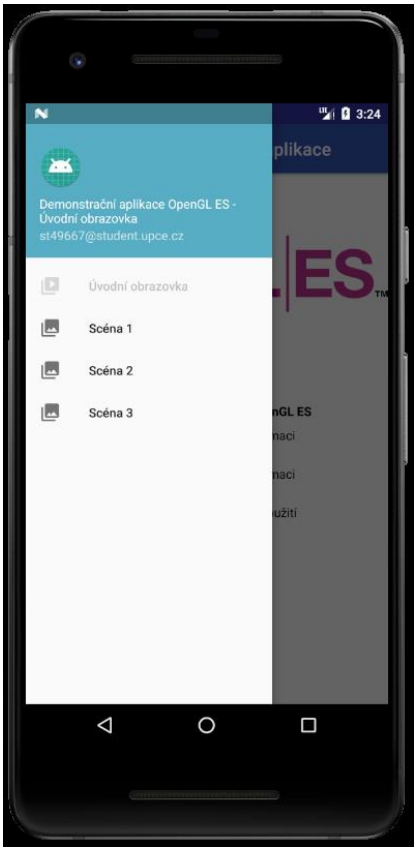
Aplikace je tvořena úvodní obrazovkou (Obrázek 5) a dalšími třemi scénami. Každá z těchto scén demonstruje některé prvky a možnosti OpenGL ES verze 2.0. Scéna 1 je 2D a scény 2 a 3 jsou 3D. Vzhled aplikace je velmi jednoduchý.

Obrázek 5: Ukázka úvodní obrazovky aplikace

Na každé scéně se nachází oblast pro vykreslování grafiky a případně nějaké ovládací prvky, které nějakým způsobem ovlivňují to, co se vykresluje ve scéně. Ze základních stavebních

komponent aplikace obsahuje pouze aktivity, žádné jiné základní komponenty nevyužívá. Na následujícím obrázku je ukázka úvodní obrazovky aplikace.

Co se týká uživatelského rozhraní, tak aplikace využívá tzv. Constraint Layout, který je v současné době doporučován pro vývoj a je také výchozím layoutem při vytvoření různých aktivit. Kromě constraint layoutu aktivity také využívají tzv. *Navigation drawer*. Navigation drawer je panel uživatelského rozhraní, který slouží k zobrazování hlavního navigačního menu aplikace (Obrázek 6). Ve výchozím stavu je schovaný, ale zobrazí se, když uživatel přejede po displeji prstem od levého okraje k pravému, nebo když klikne na ikonku v levém rohu lišty, která se nachází na horním okraji displeje. Na panelu se nachází menu, které obsahuje položky sloužící k přechodu na jednotlivé scény nebo úvodní obrazovku. Aplikace je tvořena několika třídami a tyto třídy jsou rozděleny do několika balíčků.



Obrázek 6: Ukázka navigačního menu

4.2 Třídy společné pro všechny scény

Při tvorbě demonstrační aplikace jsem nejdříve vytvářel několik tříd, které zjednodušují některé úkony spojené s OpenGL a jsou univerzální, použitelné v různých aplikacích. Tyto třídy jsem

následně využil při tvorbě jednotlivých scén. Dále jsem při tvorbě jednotlivých scén vytvářel různé specifické třídy použitelné jen v dané konkrétní situaci.

Třída `Constants`

Tato třída představuje jednotné místo pro definici různých konstant, které jsou používány na různých místech v aplikaci. Díky ní nejsou konstanty různě roztroušeny mezi různými třídami, ale všechny ostatní třídy přistupují ke konstantám, které potřebují pomocí této třídy.

Třída `VertexArray`

Aby byla práce s vertexy jednodušší, je vytvořena třída `VertexArray`. Tato třída slouží jako obálka na vertex buffer, obsahuje atribut typu `FloatBuffer`, pro ukládání dat vertexů a jednu metodu, která usnadňuje nastavení vertex bufferu pro OpenGL. Metoda `setVertexAttribPointer()` se zadanými parametry volá postupně metody `glVertexAttribPointer()` a `glEnableVertexAttribArray()`.

Třída `ShaderHelper`

Toto je pomocná třída usnadňující provádění jednotlivých operací potřebných pro načtení, nastavení a použití shaderů. Základní metodou této třídy je metoda `readShaderFromResourceFile()`, která umožňuje načíst zdrojový kód shaderu ze zdroje a vrátit ho jako textový řetězec. Tato metoda pomáhá lepší struktuře projektu, protože zdrojové kódy shaderů nemusí být napevno uloženy v textových řetězcích uvnitř nějaké třídy. Místo toho je ve složce se zdroji (`res`) vytvořena podsložka `raw`, která obsahuje soubory s příponou `.glsl`, které obsahují zdrojové kódy jednotlivých shaderů používaných v aplikaci. Další metody této třídy jako například `compileShader()`, `linkProgram()` nebo `buildProgram()` obalují volání metod třídy `GLES20`, které slouží k nastavení shaderů.

Třída `MatrixHelper`

Obsahuje jedinou metodu `perspectiveM()`, která slouží k vytvoření projekční matice. Metoda na základě zadaných parametrů vytvoří požadovanou projekční matici, která se uloží do matice zadávané jako první parametr. Pro tvorbu projekční matice lze v současné době použít metodu `perspectiveM()` ve třídě Matrix, ale ta je dostupná až od Androidu verze Ice Cream Sandwich [1]. Nicméně já se rozhodl použít vlastní třídu `MatrixHelper`.

Třída `TextureHelper`

Třída podobná třídě `ShaderHelper` usnadňuje provádění jednotlivých operací pro načtení a nastavení textur. Třída obsahuje jedinou metodu `loadTextureFromResourceFile()`, která umožňuje načíst obrázek ze zdrojů a použít ho jako texturu. Metoda provádí načtení obrázku, vytvoření textury, překopírování obrazových dat pomocí metody `textImage2D()` ze třídy `GLUtils` a vrací ID vytvořené textury. Jednotlivé obrázky používané v aplikaci jako textury jsou uloženy ve složce `drawable`, která je podsložkou složky se zdroji a tyto obrázky jsou ve formátu PNG.

Třída `Geometry`

Tato třída slouží jako obálka pro několik staticky vnořených tříd, které se používají v jiných třídách pro tvorbu různých objektů, které jsou pomocí OpenGL vykreslovány. Ve třídě se konkrétně nacházejí třídy `Point2D`, `Point3D` a `Vector2D`.

Třída `NormalizedColor`

Třída `NormalizedColor` představuje barvu ve formátu, se kterým pracuje OpenGL. OpenGL pracuje s barvami v normalizované podobě, takže hodnota nějaké osmibitové barevné složky není celé číslo v rozmezí `0-255`, ale desetinné číslo v rozmezí `0.0-1.0`. Třída tedy obsahuje tři atributy typu `float` pro uchování jednotlivých barevných složek. V místech, kde by byli potřeba tři různé proměnné, nebo pole o třech prvcích se použije tato třída a dle potřeby se přistupuje k jejím atributům. Součástí třídy je také statická metoda `generateRandomColorInRange()`, která slouží k vygenerování náhodné barvy s hodnotami barevných složek v zadaném rozsahu.

Třída `ShaderProgram`

Třída představující OpenGL program, který je tvořen dvojící vertex shader a fragment shader. Obsahuje atribut `programId`, ve kterém je uloženo ID získané pomocí metody `buildProgram()` ze třídy `ShaderHelper`. Třída obsahuje metodu `useProgram()`, ve které se pomocí volání `glUseProgram(programId)` nastaví daný program jako aktuální, který se bude používat při vykreslování. V průběhu vykreslování se může měnit, který z různých OpenGL programů bude jako aktivní, aby se dosáhlo toho, že se použijí různé shadery pro vykreslení jednotlivých objektů. Tato třída je rodičem pro další třídy, které jsou používané v jednotlivých scénách a ty tuto třídu rozšiřují o možnosti nastavení různých konstant a proměnných ve shaderech využívaných těmito scénami.

4.3 Třídy specifické pro jednotlivé scény

V této kapitole budou popsány jednotlivé třídy, které byly vytvořeny speciálně pro použití v jedné nebo více konkrétních scén.

Třídy pro aktivity

Každá scéna je tvořena jednou aktivitou a tato aktivita obsahuje plochu pro kreslení plus nějaké další komponenty využívané v dané konkrétní scéně. Jsou to třídy `SceneOneActivity`, `SceneTwoActivity` a `SceneThreeActivity` a pro úvodní obrazovku aplikace ještě třída `StartActivity`. Všechny tyto třídy jsou umístěny v základním balíčku aplikace, což je balíček `cz.upce.st49967.main`.

Třídy pro kreslicí plochy

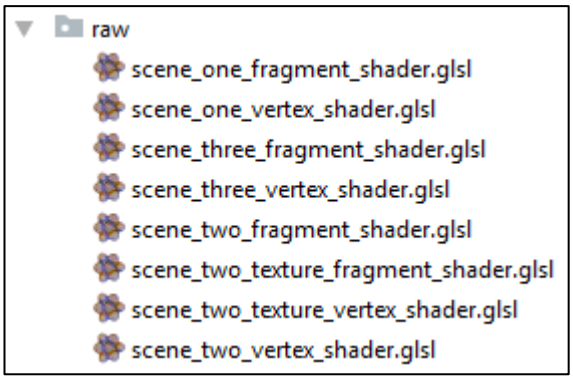
Jednotlivé scény mají každá vlastní třídu pro kreslení a všechny tyto třídy jsou potomky třídy `GLSurfaceView`. Jsou to třídy `SceneOneSurfaceView`, `SceneTwoSurfaceView` a třída `SceneThreeSurfaceView` a všechny tyto třídy se nachází v balíčku `views`.

Třídy pro renderery

Každá třída představující kreslicí plochu určité scény využívá vlastní třídu představující renderer, který se stará o vykreslování na tuto plochu. Všechny tyto třídy implementují rozhraní `GLSurfaceView.`*`Renderer`* a jeho metody `onSurfaceCreate()`, `onSurfaceChanged()` a `onDrawFrame()`, pomocí kterých jsou definovány vzhled a chování dané scény. Jsou to třídy `SceneOneRenderer`, `SceneTwoRenderer` a `SceneThreeRenderer`, které se nacházejí v balíčku `renderers`.

Třídy pro shadery (OpenGL programy)

Dále má každá scéna třídu představující OpenGL program s danými shadery. Tyto třídy jsou potomky základní třídy `ShaderProgram` a obsahují různé metody pro nastavování konstant a jiných hodnot používaných ve shaderech. Scéna 1 a scéna 3 využívají každá jednu třídu, a jsou to třídy `SceneOneShaderProgram` a `SceneThreeShaderProgram` a scéna 2 využívá třídy `SceneTwoShaderProgram` a `SceneTwoTextureShaderProgram`, které se různě při vykreslování scény střídají. Všechny tyto třídy se společně s rodičovskou třídou `ShaderProgram` nacházejí v balíčku `shaders`. Jednotlivé shadery jsou definovány jako zdroje ve složce `raw` (Obrázek 7), která se nachází uvnitř hlavní složky se zdroji (`res`).



Obrázek 7: Seznam shaderů používaných v aplikaci

Třídy představující jednotlivé objekty

Poslední skupinou tříd, jsou třídy, které představují jednotlivé objekty, které jsou vykreslovány na kreslicí povrch v jednotlivých scénách. Všechny tyto třídy se nacházejí v balíčku `objects`.

První třídou je třída `Shape2D`, která se používá ve scéně 1. Třída představuje dvojrozměrný tvar, mezi jehož atributy patří střed, který udává pozici ve 2D prostoru, velikost bodu, směrový vektor, který udává směr pohybu, kterým se tvar pohybuje, dále barva tvaru a hlavně typ tvaru. Pro typ tvaru je vytvořen výčtový typ `ShapeType` v balíčku `util`, který má hodnoty `SQUARE`, `CIRCLE`, `PENTAGON` pro tvary čtverec, kruh a pětiúhelník. Tyto tvary mohou být vykresleny ve scéně. Vykreslování tvarů se provádí pomocí textur tak, že textura obsahuje daný tvar v bílé barvě a okolí v černé barvě a při vykreslování tvarů se vykreslují ty body, které jsou v textuře bílé a použije se na ně požadovaná barva tvaru a černé body textury se nevykreslují. Obrázky, ze kterých jsou tvořeny textury tak vlastně představují jakousi masku nebo šablonu.

Další třídy jsou třídy `RectangleXZ`, `RectangleYZ` a `RectangleXY`. Tyto třídy jsou stejné, liší se jen v základní orientaci obdélníků. Například `RectangleXZ` představuje obdélník, který ve výchozím stavu leží na rovině tvořené osami X a Z. Obdélníky jsou definovány pomocí čtyř vertexů umístěných v jednotlivých rozích těchto obdélníků. Tyto třídy se používají ve scéně 2. Mezi atributy těchto tříd patří střed obdélníku, který udává pozici ve 3D prostoru, poloměr v jednom směru a poloměr ve druhém směru a barva. Obdélníky mohou být vykresleny jak pomocí jednoduché barvy, tak pomocí textur a oboje tyto možnosti jsou využívány ve scéně 2.

Třída `Cube` představuje krychli. Používá se ve scéně 2. Mezi atributy této třídy patří střed krychle ve 3D prostoru a atribut přestavující polovinu délky strany krychle. Barva jednotlivých

stěn krychle je pevně daná. Krychli lze vykreslit pomocí barev, nebo pomocí textur. Při vykreslování textur je možnost vybrat textury, které se aplikují na jednotlivé stěny krychle.

Poslední třídou je třída `PlaneXZ`. Tato třída je podobná třídě `RectangleXZ` s tím rozdílem, že obsahuje kromě čtyř vertexů v rozích ještě jeden vertex ve středu. Třída může být vykreslena pomocí jednoduché barvy. Tato třída se používá ve scéně 3.

Obrázek zobrazující strukturu projektu demonstrační aplikace s jednotlivými balíčky a jejich třídami je součástí přílohy A.

4.4 Ukázka 2D, animace a použití textur (Scéna 1)

Scéna 1 demonstruje použití OpenGL ES při práci s 2D grafikou, texturami a animacemi. Na scéně se nachází několik různých objektů, mají různé tvary a různou barvu a náhodně se pohybují po displeji (Obrázek 8). Když nějaký z objektů dosáhne okraje displeje, dojde ke změně jeho pohybu tak, aby zůstal na obrazovce. Při dotyku obrazovky dochází k pozastavení, nebo opětovnému spuštění animace (pohybu) objektů.

Obrázek 8: Ukázka scény 1

Scéna 1 využívá třídy:

- `SceneOneActivity`;
- `SceneOneSurfaceView`;
- `SceneOneRenderer`;
- `SceneOneShaderProgram`;
- `Shape2D`;
- a výčtový typ `ShapeType`.

V následující části bude podrobněji popsána třída `SceneOneRenderer`, která definuje, co se ve scéně 1 zobrazí a jak to bude vypadat.

Třída obsahuje atributy představující souřadnice jednotlivých hran obrazovky, dále projekční matici, konstanty představující minimální a maximální velikost objektů, OpenGL program, dále proměnnou, která říká, jestli se mají objekty pohybovat, hešovací tabulku, která slouží k uchovávání ID textur pro jednotlivé tvary objektů, a nakonec pole objektů.

V metodě `onSurfaceCreated()` se nejdříve načtou jednotlivé textury pro tvary objektů a uloží se do hešovací tabulky. Potom se vytvoří 12 objektů, které se uloží do pole objektů. Pro každý objekt je náhodně vygenerována jeho počáteční pozice, jeho směrový vektor, barva, a nakonec typ objektu, tzn. jestli to bude čtverec, kruh nebo pětiúhelník (Zdrojový kód 3).

Metoda `onSurfaceChanged()` slouží k vypočítání poměru stran displeje na základě jeho šířky a výšky, které jsou odvozeny od aktuální orientace zařízení a na základě toho se nastaví projekční matice pro ortografickou projekci a ještě se na základě poměru stran nastaví souřadnice hran obrazovky.

V metodě `onDrawFrame()` se vykreslí jednotlivé objekty a potom se testuje jestli se mají pohybovat. Pokud se mají pohybovat, tak se aktualizuje jejich pozice a testuje se, jestli nejsou na kraji obrazovky (Zdrojový kód 2). Pokud jsou objekty na kraji obrazovky, tak se otočí jejich směrový vektor a to způsobí, že se příště budou pohybovat opačným směrem a budou se pořád nacházet na obrazovce.

```
// pokud se mají objekty pohybovat
if (animate) {
    // pohyb objektu - aktualizace souřadnic jeho středu pomocí směrového vektoru
    object.move();
    // testování, jestli objekt dosáhl levého nebo pravého okraje obrazovky,
    // pokud dosáhl okraje dochází k inverzi složky X směrového vektoru
    if (object.getX() < leftBorder + (getSizeInCoordinates(object))
            || object.getX() > rightBorder - (getSizeInCoordinates(object))) {
        object.inverseVectorX();
    }
    // testování, jestli objekt dosáhl dolního nebo horního okraje obrazovky,
    // pokud dosáhl okraje dochází k inverzi složky Y směrového vektoru
    if (object.getY() < bottomBorder + (getSizeInCoordinates(object))
            || object.getY() > topBorder - (getSizeInCoordinates(object))) {
        object.inverseVectorY();
    }
}
```

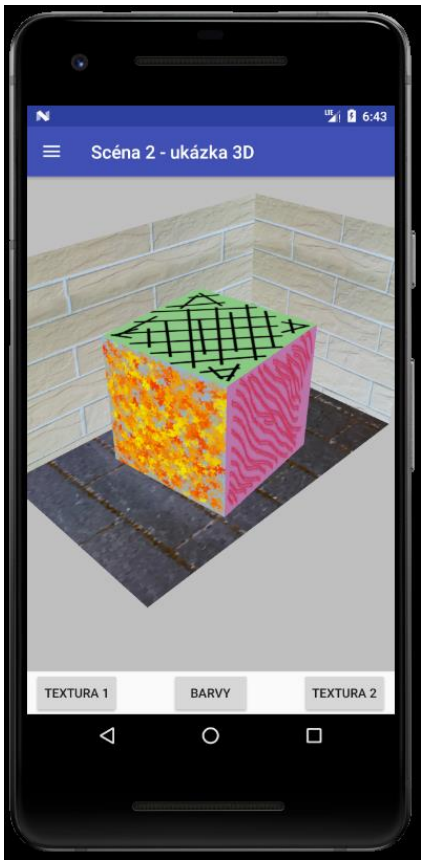
Zdrojový kód 2: Testování dosažení okrajů obrazovky

```
// generování složek počáteční pozice objektu
float randomX = generateRandomFloatInRange(rng,  min: -1.0f,  max: 1.0f);
float randomY = generateRandomFloatInRange(rng,  min: -1.0f,  max: 1.0f);
// generování složek směrového vektoru
float randomVectorX = generateRandomFloatInRange(rng,  min: 0.001f,  max: 0.012f);
float randomVectorY = generateRandomFloatInRange(rng,  min: 0.001f,  max: 0.012f);
// barvy se generují od 0.5+, aby to byly světlé barvy
float randomR = generateRandomFloatInRange(rng,  min: 0.5f,  max: 1.0f);
float randomG = generateRandomFloatInRange(rng,  min: 0.5f,  max: 1.0f);
float randomB = generateRandomFloatInRange(rng,  min: 0.5f,  max: 1.0f);
```

Zdrojový kód 3: Generování náhodných vlastností objektů

4.5 Ukázka 3D, animace a použití více různých textur (Scéna 2)

Scéna 2 demonstruje stejně jako scéna 1 práci s texturami a animacemi, ale tentokrát ve 3D. Na scéně se nachází krychle a tři obdélníky (Obrázek 9). Jeden obdélník představuje jakousi podložku, na které se nachází krychle a další dva obdélníky tvoří dvě stěny této podložky. Krychle se otáčí okolo osy Y. Jednotlivé obdélníky jsou vykresleny pomocí textur a krychle je vykreslena buď pomocí barev, nebo pomocí dvou sad textur. Ve scéně se nachází tři tlačítka pro nastavení toho, jak se bude vykreslovat krychle.



Obrázek 9: Ukázka scény 2

V této scéně jsou použity třídy:

- `SceneTwoActivity`;
- `SceneTwoSurfaceView`;
- `SceneTwoRenderer`;
- `SceneTwoShaderProgram`;
- `SceneTwoTextureShaderProgram`;
- `RectangleXZ`;
- `RectangleYZ`;
- `RectangleXY`;
- `Cube`.

V následující části bude popsána třída `SceneTwoRenderer`, která definuje jednotlivé objekty ve scéně 2.

Třída obsahuje atributy, které slouží k uložení ID jednotlivých textur, atributy pro jednotlivé objekty, dále projekční, rotační matice a matici pro nastavení pozice kamery, pomocí které se díváme na scénu. Ještě třída obsahuje výčtový typ `CubeSurfaceType`, který určuje, jestli se krychle bude vykreslovat pomocí 1. nebo 2. sady textur, nebo pomocí barev.

V metodě `onSurfaceCreated()` se nejdříve načtou všechny používané textury a uloží se do proměnných pro ně určených. Následně se vytvoří jednotlivé objekty, které se nachází ve scéně.

Metoda `onSurfaceChanged()` slouží k nastavení matice pro perspektivní projekci v závislosti na aktuální orientaci zařízení a velikosti obrazovky a také pro nastavení pozice kamery.

V metodě `onDrawFrame()` se provádí vykreslování jednotlivých objektů. Nejdříve jsou vykresleny jednotlivé obdélníky pomocí textur. Potom je provedena aktualizace rotační matice (Zdrojový kód 4), kde úhel rotace je vypočítán na základě aktuálního času a matice je potom použita pro vykreslení krychle [5]. Krychle je na základě hodnoty výčtového typu vykreslena pomocí textur nebo barvy.

```
// výpočet aktuálního úhlu rotace
long time = SystemClock.uptimeMillis() % 4000L;
float angle = 0.090f * ((int) time);
// nastavení matice rotace s vypočítaným úhlem a rotací okolo osy Y
Matrix.setRotateM(rotationMatrix, rmOffset: 0, angle, x: 0f, y: 1f, z: 0f);
```

Zdrojový kód 4: Nastavení úhlu rotace krychle

4.6 Ukázka zdroje světla (Scéna 3)

Scéna 3 demonstruje použití zdroje světla. Na scéně se nachází obdélník, který je osvětlován bodovým světlem (Obrázek 10). Ve scéně se nachází tlačítko, které umožňuje přepínat mezi dvěma barvami světla a další tři tlačítka, které umožňují přepínat mezi třemi různými pozicemi světla. Potom je zde ještě posuvník, který slouží ke změně intenzity světla.

Obrázek 10: Ukázka scény 3

Ve třetí scéně jsou použity třídy:

- `SceneThreeActivity`;
- `SceneThreeSurfaceView`;
- `SceneThreeRenderer`;
- `SceneThreeShaderProgram`;
- `PlaneXZ`.

V následující části bude podrobněji popsána třída `SceneThreeRenderer`, která se stará o vykreslování třetí scény.

Třída obsahuje atributy pro jednotlivé pozice světla, pro jednotlivé barvy světla, dále pro transformační matice, potom atribut pro aktuální intenzitu světla a atribut pro obdélník, který se bude vykreslovat.

V Metodě `onSurfaceCreated()` se pouze vytváří obdélník, který bude osvětlován zdrojem světla. Metoda `onSurfaceChanged()` slouží k nastavení matice pro perspektivní projekci a matice pro nastavení pozice kamery.

V metodě `onDrawFrame()` se pouze vykreslí obdélník, který má být vykreslen. Samotné osvětlení obdélníku na základě pozice a barvy zdroje světla se provádí uvnitř vertex shaderu (Zdrojový kód 5). Pro každý vertex zpracovávaný shaderem se vypočítává jeho barva. Barva pro fragmenty, které se nachází mezi jednotlivými vertexy bude lineárně interpolována. Barva vertexu závisí na pozici světla v prostoru, vzdálenosti světla od vertexu, na úhlu, pod kterým světlo dopadá na obdélník a také na aktuální intenzitě světla [4]. S ohledem na tyto vlastnosti se vypočítá výsledná barva světla dopadajícího na daný vertex a tato barva se přidá k původní barvě materiálu, kterou daný vertex měl. V tomto konkrétním případě je obdélník černý, takže barva materiálu je černá.

```
vec3 calculatePointLighting(){
    // 0. výchozí nastavení barvy světla na černou
    vec3 lighting = vec3(0.0);
    // 1. vypočítání vektoru od zdroje světla k vertexu na daném povrchu
    vec3 vectorToPointLight  = vec3(eyeSpacePointLightPosition) - vec3(modelViewVertex);
    // 2. vypočítání vzdálenosti zdroje světla od povrchu
    float distance = length(vectorToPointLight);
    // 3. normalizace vektoru od zdroje světla
    vectorToPointLight = normalize(vectorToPointLight);
    // 4. výpočet 'Lambertova faktoru', který závisí na úhlu dopadu světla na povrch
    float cosine = max(dot(modelViewNormal, vectorToPointLight), 0.0);
    // 5. výpočet konečné barvy, v závislosti na původní barvě, barvě světla,
    // úhlu pod kterým světlo dopadá na povrch a vzdálenosti světla od povrchu
    lighting = materialColor + ((u_PointLightColor * u_LightPower * cosine)/ distance);
    // 6. vrácení výsledné barvy vertexu
    return lighting;
}
```

Zdrojový kód 5: Funkce pro výpočet osvětlení bodovým světlem v shaderu

ZÁVĚR

Cílem této bakalářské práce bylo vytvořit ukázkovou mobilní aplikaci pro operační systém Android, která bude demonstrovat některé možnosti aplikačního rozhraní OpenGL ES.

Ukázková aplikace obsahuje úvodní obrazovku a tři různé scény, demonstrující některé z možností OpenGL ES. Pro přepínání mezi jednotlivými scénami aplikace byl použit tzv. *navigation drawer*, který představuje navigační menu, které se zobrazuje v levé části obrazovky a uživatel ho může jednoduše skrývat a zobrazovat. První scéna je 2D a demonstruje použití animací a textur. Ve scéně se nachází několik objektů různého tvaru, které se náhodně pohybují a při kontaktu s krajem obrazovky svůj pohyb změní tak, aby zůstali na obrazovce. Uživatel může dotykem obrazovky pohyb objektů pozastavit nebo znovu spustit. Druhá scéna demonstruje také použití textur a animace, ale ve 3D. V této scéně se nachází rotující krychle, na kterou má uživatel možnost pomocí tlačítka aplikovat dvě různé sady textur. Třetí scéna demonstruje použití světelných zdrojů, konkrétně bodového světla. Ve scéně se nachází čtvercová plocha, která je z různých míst ozařována bodovým světlem a uživatel může přepínat jednotlivé pozice světla a také může měnit intenzitu zdroje světla.

Při tvorbě této práce jsem se naučil mnoho nových věcí z oblasti počítačové grafiky, které mohu využít i jinde než při práci s OpenGL. Do ukázkové aplikace by bylo možné v budoucnu přidat různé další efekty, nebo by se mohli některé její komponenty použít pro vytvoření nějaké mobilní hry.

POUŽITÁ LITERATURA

[1] BROTHALER, Kevin. *OpenGL ES 2 for Android: a quick-start guide*. Dallas, Texas: Pragmatic Bookshelf, 2013. ISBN 978-1-937785-34-5.

[2] GRIFFITHS, Dawn a David GRIFFITHS. *Head first Android development*. Sebastopol: O'Reilly, 2015. Head first series. ISBN 14-493-6218-4.

[3] Platform Architecture. *Android Developers* [online]. [cit. 2017-12-20]. Dostupné z: https://developer.android.com/guide/platform/index.html

[4] Android Lesson Two: Ambient and Diffuse Lighting. *Learn OpenGL ES* [online]. [cit. 2018-04-14]. Dostupné z: http://www.learnopengles.com/android-lesson-two-ambient-and-diffuse-lighting/

[5] Add motion. *Android Developers* [online]. [cit. 2018-04-28]. Dostupné z: https://developer.android.com/training/graphics/opengl/motion

[6] Android.opengl. *Android Developers* [online]. [cit. 2018-04-28]. Dostupné z: https://developer.android.com/reference/android/opengl/package-summary

PŘÍLOHY

PŘÍLOHA A – STRUKTURA PROJEKTU DEMONSTRAČNÍ APLIKACE

Následující obrázek zobrazuje strukturu projektu aplikace, jednotlivých balíčků a jejich tříd.